

6

Grafos dirigidos

En los problemas originados en ciencias de la computación, matemáticas, ingeniería y muchas otras disciplinas, a menudo es necesario representar relaciones arbitrarias entre objetos de datos. Los grafos dirigidos y los no dirigidos son modelos naturales de tales relaciones. Este capítulo presenta las estructuras de datos básicas que pueden usarse para representar grafos dirigidos. También se presentan algunos algoritmos básicos para la determinación de conectividad en grafos dirigidos y para encontrar los caminos más cortos.

6.1 Definiciones fundamentales

Un *grafo dirigido* G consiste en un conjunto de vértices V y un conjunto de arcos A . Los vértices se denominan también *nodos* o *puntos*; los arcos pueden llamarse *arcos dirigidos* o *líneas dirigidas*. Un arco es un par ordenado de vértices (v, w) ; v es la *cola* y w la *cabeza* del arco. El arco (v, w) se expresa a menudo como $v \rightarrow w$ y se representa como



Obsérvese que la «punta de la flecha» está en el vértice llamado «cabeza», y la cola, en el vértice llamado «cola». Se dice que el arco $v \rightarrow w$ va de v a w , y que w es *adyacente* a v .

Ejemplo 6.1. La figura 6.1 muestra un grafo dirigido con cuatro vértices y cinco arcos. □

Los vértices de un grafo dirigido pueden usarse para representar objetos, y los arcos, relaciones entre los objetos. Por ejemplo, los vértices pueden representar ciudades, y los arcos, vuelos aéreos de una ciudad a otra. En otro ejemplo, como el que se presentó en la sección 4.2, un grafo dirigido puede emplearse para representar el flujo de control en un programa de computador. Los vértices representan bloques básicos, y los arcos, posibles transferencias del flujo de control.

Un *camino* en un grafo dirigido es una secuencia de vértices v_1, v_2, \dots, v_n , tal que $v_1 \rightarrow v_2, v_2 \rightarrow v_3, \dots, v_{n-1} \rightarrow v_n$ son arcos. Este camino va del vértice v_1 al vértice v_n .

pasa por los vértices v_2, v_3, \dots, v_{n-1} , y termina en el vértice v_n . La *longitud* de un camino es el número de arcos en ese camino, en este caso, $n - 1$. Como caso especial, un vértice sencillo, v , por sí mismo denota un camino de longitud cero de v a v . En la figura 6.1, la secuencia 1, 2, 4 es un camino de longitud 2 que va del vértice 1 al vértice 4.

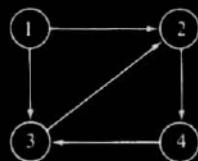


Fig. 6.1. Grafo dirigido.

Un camino es *simple* si todos sus vértices, excepto tal vez el primero y el último, son distintos. Un *ciclo simple* es un camino simple de longitud por lo menos uno, que empieza y termina en el mismo vértice. En la figura 6.1, el camino 3, 2, 4, 3 es un ciclo de longitud 3.

En muchas aplicaciones es útil asociar información a los vértices y arcos de un grafo dirigido. Para este propósito es posible usar un *grafo dirigido etiquetado*, en el cual cada arco, cada vértice o ambos pueden tener una etiqueta asociada. Una etiqueta puede ser un nombre, un costo o un valor de cualquier tipo de datos dado.

Ejemplo 6.2. La figura 6.2 muestra un grafo dirigido etiquetado en el que cada arco está etiquetado con una letra que causa una transición de un vértice a otro. Este grafo dirigido etiquetado tiene la interesante propiedad de que las etiquetas de los arcos de cualquier ciclo que sale del vértice 1 y vuelve a él producen una cadena de caminos a y b en la cual los números de a y de b son pares. \square

En un grafo dirigido etiquetado, un vértice puede tener a la vez un nombre y una etiqueta. A menudo se empleará la etiqueta del vértice como si fuera el nombre. Así, los números de la figura 6.2 pueden interpretarse como nombres o como etiquetas de vértices.

6.2 Representaciones de grafos dirigidos

Para representar un grafo dirigido se pueden emplear varias estructuras de datos; la selección apropiada depende de las operaciones que se aplicarán a los vértices y a los arcos del grafo. Una representación común para un grafo dirigido $G = (V, A)$ es la *matriz de adyacencia*. Supóngase que $V = \{1, 2, \dots, n\}$. La matriz de adyacencia para G es una matriz A de dimensión $n \times n$, de elementos booleanos, donde $A[i, j]$ es verdadero si, y sólo si, existe un arco que vaya del vértice i al j . Con frecuencia se exhibirán matrices de adyacencias con 1 para verdadero y 0 para falso; las matrices de adyacencias pueden incluso obtenerse de esa forma. En la representación con una

matriz de adyacencia, el tiempo de acceso requerido a un elemento es independiente del tamaño de V y A . Así, la representación con matriz de adyacencia es útil en los algoritmos para grafos, en los cuales suele ser necesario saber si un arco dado está presente.

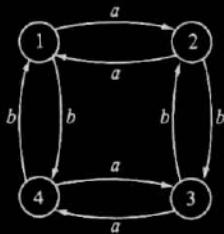


Fig. 6.2. Grafo dirigido de transiciones.

Algo muy relacionado con esto es la representación con *matriz de adyacencia etiquetada* de un grafo dirigido, donde $A[i, j]$ es la etiqueta del arco que va del vértice i al vértice j . Si no existe un arco de i a j , debe emplearse como entrada para $A[i, j]$ un valor que no pueda ser una etiqueta válida.

Ejemplo 6.3 La figura 6.3. muestra la matriz de adyacencia etiquetada para el grafo dirigido de la figura 6.2. Aquí, el tipo de la etiqueta es un carácter, y un espacio representa la ausencia de un arco. \square

	1	2	3	4
1		a	b	
2	a		b	
3		b		a
4	b		a	

Fig. 6.3. Matriz de adyacencia etiquetada para el grafo dirigido de la figura 6.2.

La principal desventaja de usar una matriz de adyacencia para representar un grafo dirigido es que requiere un espacio $\Omega(n^2)$ aun si el grafo dirigido tiene menos de n^2 arcos. Sólo leer o examinar la matriz puede llevar un tiempo $O(n^2)$, lo cual invalidaría los algoritmos $O(n)$ para la manipulación de grafos dirigidos con $O(n)$ arcos.

Para evitar esta desventaja, se puede utilizar otra representación común para un grafo dirigido $G = (V, A)$ llamada representación con *lista de adyacencia*. La lista de adyacencia para un vértice i es una lista, en algún orden, de todos los vértices adyacentes a i . Se puede representar G por medio de un arreglo *CABEZA*, donde *CABEZA*[i] es un apuntador a la lista de adyacencia del vértice i . La representación con lista de adyacencia de un grafo dirigido requiere un espacio proporcional a la suma del número de vértices más el número de arcos; se usa bastante cuando

el número de arcos es mucho menor que n^2 . Sin embargo, una desventaja potencial de la representación con lista de adyacencia es que puede llevar un tiempo $\mathcal{O}(n)$ determinar si existe un arco del vértice i al vértice j , ya que puede haber $\mathcal{O}(n)$ vértices en la lista de adyacencia para el vértice i .

Ejemplo 6.4. La figura 6.4 muestra una representación con lista de adyacencia para el grafo dirigido de la figura 6.1, donde se usan listas enlazadas sencillas. Si los arcos tienen etiquetas, éstas podrían incluirse en las celdas de la lista ligada.

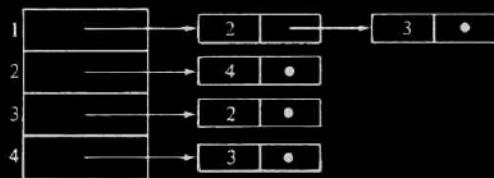


Fig. 6.4. Representación con lista de adyacencia para el grafo dirigido de la figura 6.1

Si hubo inserciones y supresiones en las listas de adyacencias, sería preferible tener el arreglo *CABEZA* apuntando a celdas de encabezamiento que no contienen vértices adyacentes †. Por otra parte, si se espera que el grafo permanezca fijo, sin cambios (o con muy pocos) en las listas de adyacencias, sería preferible que *CABEZA*[*i*] fuera un cursor a un arreglo *ADY*, donde *ADY*[*CABEZA*[*i*]], *ADY*[*CABEZA*[*i*] + 1], ..., y así sucesivamente, contuvieran los vértices adyacentes al vértice *i*, hasta el punto en *ADY* donde se encuentra por primera vez un cero, el cual marca el fin de la lista de vértices adyacentes a *i*. Por ejemplo, la figura 6.1 puede representarse con la figura 6.5. □

TDA grafo dirigido

Se podría definir un TDA que correspondiera formalmente al grafo dirigido y estudiar las implantaciones de sus operaciones. No se redundará demasiado en esto, porque hay poco material en verdad nuevo y las principales estructuras de datos para grafos ya han sido cubiertas. Las operaciones más comunes en grafos dirigidos incluyen la lectura de la etiqueta de un vértice o un arco, la inserción o supresión de vértices y arcos, y el recorrido de arcos desde la cola hasta la cabeza.

Las últimas operaciones requieren más cuidado. Con frecuencia, se encuentran en proposiciones informales de programas como

for cada vértice *w* adyacente al vértice *v* do
{|alguna acción sobre *w*|} (6.1)

† Esta es otra manifestación del viejo problema de Pascal de hacer inserción y supresión en posiciones arbitrarias de listas enlazadas sencillas.

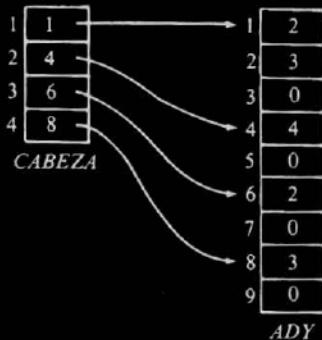


Fig. 6.5. Otra representación con lista de adyacencia de la figura 6.1.

Para obtener esto, es necesaria la noción de un tipo *índice* para el conjunto de vértices adyacentes a algún vértice v . Por ejemplo, si las listas de adyacencias se usan para representar el grafo, un índice es, en realidad, una posición en la lista de adyacencia de v . Si se usa una matriz de adyacencia, un índice es un entero que representa un vértice adyacente. Se requieren las tres operaciones siguientes en grafos dirigidos.

1. PRIMERO(v), que devuelve el índice del primer vértice adyacente a v . Se devuelve un vértice nulo Λ si no existe ningún vértice adyacente a v .
2. SIGUIENTE(v, i), que devuelve el índice posterior al índice i de los vértices adyacentes a v . Se devuelve Λ si i es el último vértice de los vértices adyacentes a v .
3. VERTICE(v, i), que devuelve el vértice cuyo índice i está entre los vértices adyacentes a v .

Ejemplo 6.5. Si se escoge la representación con matriz de adyacencia, VERTICE(v, i) devuelve i . PRIMERO(v) y SIGUIENTE(v, i) pueden escribirse como en la figura 6.6, para operar en una matriz booleana A de $n \times n$ definida de manera externa. Se supone que A está declarada como

```
array [1..n, 1..n] of boolean
```

y que 0 se emplea para Λ . Después, se obtiene la proposición (6.1) como en la figura 6.7. \square

```
function PRIMERO ( v: integer) : integer;
var
  i: integer;
```

```

begin
  for i := 1 to n do
    if A[v, i] then
      return(i);
    return (0) { si se llega aquí, v no tiene vértices adyacentes }
  end; { PRIMERO }

function SIGUIENTE ( v: integer; i: integer ) : integer;
var
  j: integer;
begin
  for j := i+1 to n do
    if A[v, j] then
      return (j);
  return (0)
end; { SIGUIENTE }

```

Fig. 6.6. Operaciones para recorrer vértices adyacentes.

```

i := PRIMERO(v);
while i <>^ do begin
  w := VERTICE(v, i);
  { alguna acción en w }
  i := SIGUIENTE(v, i)
end

```

Fig. 6.7. Iteración en vértices adyacentes a v.

6.3 Problema de los caminos más cortos con un solo origen

En esta sección se considera un problema común de búsqueda de caminos en grafos dirigidos. Supóngase un grafo dirigido $G = (V, A)$ en el cual cada arco tiene una etiqueta no negativa, y donde un vértice se especifica como *origen*. El problema es determinar el costo del camino más corto del origen a todos los demás vértices de V , donde la *longitud de un camino* es la suma de los costos de los arcos del camino. Esto se conoce con el nombre de problema de *los caminos más cortos con un solo origen* †. Obsérvese que se hablará de caminos con «longitud» aun cuando los costos representan algo diferente, como tiempo.

Sea G un mapa de vuelos en el cual cada vértice representa una ciudad, y cada

† Se puede esperar que un problema más natural sea encontrar el camino más corto entre el origen y un vértice *destino* particular. Sin embargo, ese problema parece tan difícil en general como el de los caminos más cortos con un solo origen (a menos que se tenga la suerte de encontrar el camino al destino antes que alguno de los otros vértices y así terminar el algoritmo un poco antes que si se buscaran los caminos hacia todos los vértices).

arco $v \rightarrow w$, una ruta aérea de la ciudad v a la ciudad w . La etiqueta del arco $v \rightarrow w$ es el tiempo que se requiere para volar de v a w [†]. La solución del problema de los caminos más cortos con un solo origen para este grafo dirigido determinaría el tiempo de viaje mínimo para ir de cierta ciudad a todas las demás del mapa.

Para resolver este problema se manejará una técnica «ávida» conocida como *algoritmo de Dijkstra*, que opera a partir de un conjunto S de vértices cuya distancia más corta desde el origen ya es conocida. En principio, S contiene sólo el vértice de origen. En cada paso, se agrega algún vértice restante v a S , cuya distancia desde el origen es la más corta posible. Suponiendo que todos los arcos tienen costo no negativo, siempre es posible encontrar un camino más corto entre el origen y v que pasa sólo a través de los vértices de S , y que se llama *especial*. En cada paso del algoritmo, se utiliza un arreglo D para registrar la longitud del camino especial más corto a cada vértice. Una vez que S incluye todos los vértices, todos los caminos son «especiales», así que D contendrá la distancia más corta del origen a cada vértice.

El algoritmo se da en la figura 6.8, donde se supone que existe un grafo dirigido $G = (V, A)$ en el que $V = \{1, 2, \dots, n\}$ y el vértice 1 es el origen. C es un arreglo bidimensional de costos, donde $C[i, j]$ es el costo de ir del vértice i al vértice j por el arco $i \rightarrow j$. Si no existe el arco $i \rightarrow j$, se supone que $C[i, j] = \infty$, un valor mucho mayor que cualquier costo real. En cada paso, $D[i]$ contiene la longitud del camino especial más corto actual para el vértice i .

Ejemplo 6.6. Aplíquese *Dijkstra* al grafo dirigido de la figura 6.9. En principio, $S = \{1\}$, $D[2] = 10$, $D[3] = \infty$, $D[4] = 30$ y $D[5] = 100$. En la primera iteración del ciclo **for** de las líneas (4) a (8), $w = 2$ se selecciona como el vértice con el mínimo valor D . Después se hace $D[3] = \min(\infty, 10 + 50) = 60$. $D[4]$ y $D[5]$ no cambian porque el camino para llegar a ellos directamente desde 1 es más corto que pasar por el vértice 2. La secuencia de valores D después de cada iteración se muestra en la figura 6.10. □

Para reconstruir el camino más corto del origen a cada vértice, se agrega otro arreglo P de vértices, tal que $P[v]$ contenga el vértice inmediato anterior a v en el camino más corto. Se asigna $P[v]$ valor inicial 1 para toda $v \neq 1$. El arreglo P puede actualizarse después de la línea (8) de *Dijkstra*. Si $D[w] + C[w, v] < D[v]$ en la línea (8), después se hace $P[v] := w$. Al término de *Dijkstra*, el camino a cada vértice puede de encontrarse regresando por los vértices predecesores del arreglo P .

```

procedure Dijkstra;
  { Dijkstra calcula el costo de los caminos más cortos entre el vértice 1
    y todos los demás de un grafo dirigido }
begin
  (1)   S := { 1 };
  (2)   for i := 2 to n do

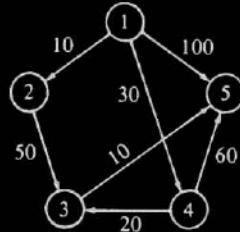
```

[†] Cabría suponer que puede usarse un grafo no dirigido, ya que la etiqueta de los arcos $v \rightarrow w$ y $w \rightarrow v$ sería la misma. Sin embargo, los tiempos de viaje son diferentes en direcciones diferentes, debido a los vientos. De cualquier forma, aunque las etiquetas $v \rightarrow w$ y $w \rightarrow v$ fueran idénticas, esto no ayudaría a resolver el problema.

```

(3)       $D[i] := C[1, i]; \{ \text{asigna valor inicial a } D \}$ 
(4)      for  $i := 1$  to  $n-1$  do begin
(5)          elige un vértice  $w$  en  $V-S$  tal que  $D[w]$  sea un mínimo;
(6)          agrega  $w$  a  $S$ ;
(7)          for cada vértice  $v$  en  $V-S$  do
(8)               $D[v] := \min(D[v], D[w] + C[w, v])$ 
end
end;  $\{ \text{Dijkstra} \}$ 

```

Fig. 6.8. Algoritmo de Dijkstra.**Fig. 6.9.** Grafo dirigido con arcos etiquetados.

Ejemplo 6.7. Para el grafo dirigido del ejemplo 6.6, el arreglo P debe tener los valores $P[2] = 1$, $P[3] = 4$, $P[4] = 1$ y $P[5] = 3$. Para encontrar el camino más corto del vértice 1 al vértice 5, por ejemplo, se siguen los predecesores en orden inverso comenzando en 5. A partir del arreglo P , se determina que 3 es el predecesor de 5, 4 el predecesor de 3 y 1 el predecesor de 4. Así, el camino más corto entre los vértices 1 y 5 es 1, 4, 3, 5. \square

Iteración	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
inicial	{1}	—	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Fig. 6.10. Cálculos de Dijkstra en el grafo dirigido de la figura 6.9.

Por qué funciona el algoritmo de Dijkstra

El algoritmo de Dijkstra es un ejemplo donde la «avidez» funciona, en el sentido de que lo que aparece localmente como lo mejor, se convierte en lo mejor de todo. En este caso, lo «mejor» localmente es encontrar la distancia al vértice w que

está fuera de S , pero tiene el camino especial más corto. Para ver por qué en este caso no puede haber un camino no especial más corto desde el origen hasta w , obsérvese la figura 6.11, en la que se muestra un camino hipotético más corto a w que primero sale de S para ir al vértice x , después (tal vez) entra y sale de S varias veces antes de llegar finalmente a w .

Pero si este camino es más corto que el camino especial más corto a w , el segmento inicial del camino entre el origen y x es un camino especial a x más corto que el camino especial más corto a w . (Obsérvese lo importante que es el hecho de que los costos no sean negativos; sin ello, este argumento no sería válido, y de hecho, el algoritmo de Dijkstra no funcionaría correctamente.) En este caso, se debió seleccionar x en vez de w en la línea (5) de la figura 6.8, porque $D[x]$ fue menor que $D[w]$.

Para completar la demostración de que la figura 6.8 funciona, se verifica que en todos los casos $D[v]$ es realmente la distancia más corta de un camino especial al vértice v . La clave de este razonamiento está en observar que al agregar un nuevo vértice w a S en la línea (6), las líneas (7) y (8) ajustan D para tener en cuenta la posibilidad de que exista ahora un camino especial más corto a v a través de w . Si ese camino va a través del anterior S a w , e inmediatamente después a v , su costo, $D[w] + C[w, v]$, será comparado con $D[v]$ en la línea (8), y $D[v]$ se reducirá si el nuevo camino especial es más corto. La otra posibilidad de un camino especial más corto se muestra en la figura 6.12, donde el camino va a w , después regresa al anterior S , a algún miembro x del S anterior, y luego a v .

Pero en realidad no puede existir tal camino; puesto que x se colocó en S antes que w , el más corto de todos los caminos entre el origen y x pasa sólo a través del S anterior. Por tanto, el camino hacia x a través de w , mostrado en la figura 6.12, no es más corto que el camino que va directo a x a través de S . Como resultado, la longitud del camino de la figura 6.12 entre el origen y w , x , y v no es menor que el anterior valor de $D[v]$, ya que $D[v]$ no fue mayor que la longitud del camino más corto hasta x a través de S y después directamente a w . Así, $D[v]$ no puede reducirse en la línea (8) por medio de un camino que pase por w y x , como el de la figura 6.12, y no es necesario considerar la longitud de esos caminos.

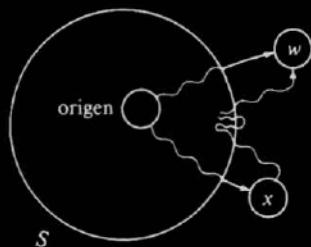


Fig. 6.11. Camino hipotético más corto a w .

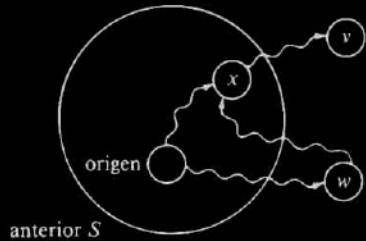


Fig. 6.12. Camino especial más corto imposible.

Tiempo de ejecución del algoritmo de Dijkstra

Supóngase que la figura 6.8 opera en un grafo dirigido con n vértices y a aristas. Si se emplea una matriz de adyacencia para representar el grafo dirigido, el ciclo de las líneas (7) y (8) lleva un tiempo $O(n)$, y se ejecuta $n - 1$ veces para un tiempo total de $O(n^2)$. El resto del algoritmo, como se puede observar, no requiere más tiempo que esto.

Si a es mucho menor que n^2 , puede ser mejor utilizar una representación con lista de adyacencia del grafo dirigido y emplear una cola de prioridad obtenida a manera de árbol parcialmente ordenado para organizar los vértices de $V - S$. El ciclo de las líneas (7) y (8) se puede realizar recorriendo la lista de adyacencia para w y actualizando las distancias en la cola de prioridad. Se hará un total de a actualizaciones, cada una con un costo de tiempo $O(\log n)$, por lo que el tiempo total consumido en las líneas (7) y (8) es ahora $O(a \log n)$, en vez de $O(n^2)$.

Las líneas (1) a (3) llevan un tiempo $O(n)$, al igual que las líneas (4) y (6). Al manejar la cola de prioridad para representar $V - S$, las líneas (5) y (6) implantan exactamente la operación SUPRIME-MIN y cada una de las $n - 1$ iteraciones de esas líneas requiere un tiempo $O(\log n)$.

Como resultado, el tiempo total consumido en esta versión del algoritmo de Dijkstra está acotado por $O(a \log n)$. Este tiempo de ejecución es mucho mejor que $O(n^2)$ si a es muy pequeña, comparada con n^2 .

6.4 Problema de los caminos más cortos entre todos los pares

Supóngase que se tiene un grafo dirigido etiquetado que da el tiempo de vuelo para ciertas rutas entre ciudades, y se desea construir una tabla que brinde el menor tiempo requerido para volar entre dos ciudades cualesquiera. Este es un ejemplo del problema de los caminos más cortos entre todos los pares (CMCP). Para plantear el problema con precisión, se emplea un grafo dirigido $G = (V, A)$ en el cual cada arco $v \rightarrow w$ tiene un costo no negativo $C[v, w]$. El problema CMCP es encontrar el camino de longitud más corta entre v y w para cada par ordenado de vértices (v, w) .

Podría resolverse este problema por medio del algoritmo de Dijkstra, tomando por turno cada vértice como vértice origen, pero una forma más directa de solución es mediante el algoritmo creado por R. W. Floyd. Por conveniencia, se supone otra vez que los vértices en v están numerados 1, 2, ..., n . El algoritmo de Floyd usa una matriz A de $n \times n$ en la que se calculan las longitudes de los caminos más cortos. Inicialmente se hace $A[i, j] = C[i, j]$ para toda $i \neq j$. Si no existe un arco que vaya de i a j , se supone que $C[i, j] = \infty$. Cada elemento de la diagonal se hace 0.

Después, se hacen n iteraciones en la matriz A . Al final de la k -ésima iteración, $A[i, j]$ tendrá por valor la longitud más pequeña de cualquier camino que vaya desde el vértice i hasta el vértice j y que no pase por un vértice con número mayor que k . Esto es, i y j , los vértices extremos del camino, pueden ser cualquier vértice, pero todo vértice intermedio debe ser menor o igual que k .

En la k -ésima iteración se aplica la siguiente fórmula para calcular A .

$$A_k[i, j] = \min \begin{cases} A_{k-1}[i, j] \\ A_{k-1}[i, k] + A_{k-1}[k, j] \end{cases}$$

El subíndice k denota el valor de la matriz A después de la k -ésima iteración; no indica la existencia de n matrices distintas. Pronto, se eliminarán esos subíndices. Esta fórmula tiene la interpretación simple de la figura 6.13.

Para obtener $A_k[i, j]$, se compara $A_{k-1}[i, j]$, el costo de ir de i a j sin pasar por k o cualquier otro vértice con numeración mayor, con $A_{k-1}[i, k] + A_{k-1}[k, j]$, el costo de ir primero de i a k y después de k a j , sin pasar a través de un vértice con número mayor que k . Si el paso por el vértice k produce un camino más económico que el de $A_{k-1}[i, j]$, se elige ese costo para $A_k[i, j]$.

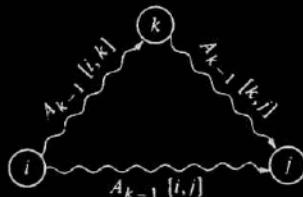


Fig. 6.13. Inclusión de k entre los vértices que van de i a j .

Ejemplo 6.8. Considérese el grafo dirigido ponderado que se muestra en la figura 6.14. En la figura 6.15 se muestran los valores iniciales de la matriz A , y después de tres iteraciones. □

Como $A_k[i, k] = A_{k-1}[i, k]$ y $A_k[k, j] = A_{k-1}[k, j]$, ninguna entrada con cualquier subíndice igual a k cambia durante la k -ésima iteración. Por tanto, se puede realizar el cálculo sólo con una copia de la matriz A . En la figura 6.16 se muestra un programa para realizar este cálculo en matrices de $n \times n$.

Es evidente que el tiempo de ejecución de este programa es $\mathcal{O}(n^3)$, ya que el programa está conformado por el triple ciclo anidado `for`. Para verificar que este progra-

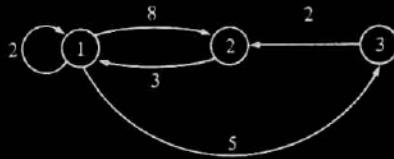


Fig. 6.14. Grafo dirigido ponderado.

ma funciona, es fácil demostrar por inducción sobre k que después de k recorridos por el triple ciclo **for**, $A[i, j]$ contiene la longitud del camino más corto desde el vértice i hasta el vértice j que no pasa a través de un vértice con número mayor que k .

	1	2	3		1	2	3	
1	0	8	5	$A_0[i, j]$	0	8	5	$A_1[i, j]$
2	3	0	∞	3	0	8		$A_2[i, j]$
3	∞	2	0	∞	2	0		$A_3[i, j]$

Fig. 6.15. Valores de matrices A sucesivas.

```

procedure Floyd ( var A: array[1..n, 1..n] of real;
                  C: array[1..n, 1..n] of real );
{ Floyd calcula la matriz A de caminos más cortos dada la matriz de costos
  de arcos C }
var
  i, j, k: integer;
begin
  for i := 1 to n do
    for j := 1 to n do
      A[i, j] := C[i, j];
  for i := 1 to n do
    A[i, i] := 0;
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        if (A[i, k] + A[k, j]) < A[i, j] then
          A[i, j] := A[i, k] + A[k, j]
end; { Floyd }

```

Fig. 6.16. Algoritmo de Floyd.

Comparación entre los algoritmos de Floyd y Dijkstra

Dado que la versión de *Dijkstra* con matriz de adyacencia puede encontrar los caminos más cortos desde un vértice en un tiempo $O(n^2)$, como el algoritmo de Floyd, también puede encontrar todos los caminos más cortos en un tiempo $O(n^3)$. El compilador, la máquina y los detalles de realización determinarán las constantes de proporcionalidad. La experimentación y medición son la forma más fácil de descubrir el mejor algoritmo para la aplicación en cuestión.

Si a , el número de aristas, es mucho menor que n^2 , aun con el factor constante relativamente bajo en el tiempo de ejecución $O(n^3)$ de *Floyd*, cabe esperar que la versión de *Dijkstra* con lista de adyacencia, tomando un tiempo $O(na \log n)$ para resolver el CMCP, sea superior, al menos para grafos grandes y poco densos.

Recuperación de los caminos

En muchos casos se desea imprimir el camino más económico entre dos vértices. Un modo de lograrlo es usando otra matriz P , donde $P[i, j]$ tiene el vértice k que permitió a *Floyd* encontrar el valor más pequeño de $A[i, j]$. Si $P[i, j] = 0$, el camino más corto de i a j es directo, siguiendo el arco entre ambos. La versión modificada de *Floyd* de la figura 6.17 almacena los vértices intermedios apropiados en P .

```

procedure más_corto ( var A: array[1..n, 1..n] of real;
                      C: array[1..n, 1..n] of real; P: array[1..n, 1..n] of integer );
  | más_corto toma una matriz de costos de arcos C de  $n \times n$  y produce una matriz A
    de  $n \times n$  de longitudes de caminos más cortos y una matriz P de  $n \times n$ 
    que da un punto en la «mitad» de cada camino más corto |

var
  i, j, k: integer;
begin
  for i := 1 to n do
    for j := 1 to n do begin
      A[i, j] := C[i, j];
      P[i, j] := 0
    end;
  for i := 1 to n do
    A[i, i] := 0;
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        if A[i, k] + A[k, j] < A[i, j] then begin
          A[i, j] := A[i, k] + A[k, j];
          P[i, j] := k
        end
    end;
end; | más_corto |

```

Fig. 6.17. Programa para los caminos más cortos.

Para imprimir los vértices intermedios del camino más corto del vértice i hasta el vértice j , se invoca el procedimiento $camino(i, j)$ dado en la figura 6.18. Mientras que en una matriz arbitraria P , $camino$ puede iterar infinitamente, si P viene del procedimiento $más_corto$, no es posible tener, por ejemplo, k en el camino más corto de i a j y también tener j en el camino más corto de i a k . Obsérvese cómo la suposición de pesos no negativos es crucial otra vez.

```
procedure camino ( i, j: integer );
  var
    k: integer;
  begin
    k := P[i, j];
    if k = 0 then
      return;
    camino(i, k);
    writeln(k);
    camino(k, j)
  end; { camino }
```

Fig. 6.18. Procedimiento para imprimir el camino más corto.

Ejemplo 6.9. La figura 6.19 muestra la matriz P final para el grafo dirigido de la figura 6.14. \square

$$\begin{array}{c|ccc}
 & 1 & 2 & 3 \\
 \hline
 1 & 0 & 3 & 0 \\
 2 & 0 & 0 & 1 \\
 3 & 2 & 0 & 0
\end{array} \quad P$$

Fig. 6.19. Matriz P para el grafo dirigido de la figura 6.14.

Cerradura transitiva

En algunos problemas podría ser interesante saber sólo si existe un camino de longitud igual o mayor que uno que vaya desde el vértice i al vértice j . El algoritmo de Floyd puede especializarse para este problema; el algoritmo resultante, que antecede al de Floyd, se conoce como algoritmo de Warshall.

Supóngase que la matriz de costo C es sólo la matriz de adyacencia para el grafo dirigido dado. Esto es, $C[i, j] = 1$ si hay un arco de i a j , y 0 si no lo hay. Se desea obtener la matriz A tal que $A[i, j] = 1$ si hay un camino de longitud igual o mayor que uno de i a j , y 0 en otro caso. A se conoce a menudo como *cerradura transitiva* de la matriz de adyacencia.

Ejemplo 6.10. La figura 6.20 muestra la cerradura transitiva para la matriz de adyacencia del grafo dirigido de la figura 6.14. \square

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Fig. 6.20. Cerradura transitiva.

La cerradura transitiva puede obtenerse con un procedimiento similar a *Floyd* aplicando la siguiente fórmula en el k -ésimo paso en la matriz booleana A .

$$A_k[i, j] = A_{k-1}[i, j] \text{ o } (A_{k-1}[i, k] \text{ y } A_{k-1}[k, j])$$

Esta fórmula establece que hay un camino de i a j que no pasa por un vértice con número mayor que k si

1. ya existe un camino de i a j que no pasa por un vértice con número mayor que $k - 1$, o si
2. hay un camino de i a k que no pasa por un vértice con número mayor que $k - 1$, y un camino de k a j que no pasa por un vértice con número mayor que $k - 1$.

Igual que antes, $A_k[i, k] = A_{k-1}[i, k]$ y $A_k[k, j] = A_{k-1}[k, j]$, así que se puede realizar el cálculo con sólo una copia de la matriz A . El programa en Pascal resultante, llamado *Warshall* por su descubridor, se muestra en la figura 6.21.

```

procedure Warshall ( var A: array[1..n, 1..n] of boolean;
                     C: array[1..n, 1..n] of boolean );
  { Warshall convierte a A en la cerradura transitiva de C }
  var
    i, j, k: integer;
  begin
    for i := 1 to n do
      for j := 1 to n do
        A[i, j] := C[i, j];
    for k := 1 to n do
      for i := 1 to n do
        for j := 1 to n do
          if A[i, j] = false then
            A[i, j] := A[i, k] and A[k, j];
  end; { Warshall }

```

Fig. 6.21. Algoritmo de Warshall para cerradura transitiva.

Un ejemplo: localización del centro de un grafo dirigido

Supóngase que se desea determinar el vértice más central de un grafo dirigido. Este problema puede resolverse fácilmente con el algoritmo de Floyd. Primero, se hace

más preciso el término «vértice más central». Sea v un vértice de un grafo dirigido $G = (V, A)$. La *excentricidad* de v es

$$\max_{w \in V} \{\text{longitud mínima de un camino de } w \text{ a } v\}$$

El *centro* de G es un vértice de mínima excentricidad. Así, el centro de un grafo dirigido es un vértice más cercano al vértice más distante.

Ejemplo 6.11. Considérese el grafo dirigido ponderado de la figura 6.22.

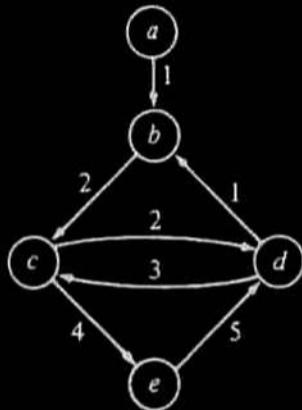


Fig. 6.22. Grafo dirigido ponderado.

Las excentricidades de los vértices son

vértice	excentricidad
a	∞
b	6
c	8
d	5
e	7

Por tanto, el centro es el vértice d . \square

Encontrar el centro de un grafo dirigido G es fácil. Supóngase que C es la matriz de costos para G .

1. Primero se aplica el procedimiento *Floyd* de la figura 6.16 a C para obtener la matriz A de los caminos más cortos entre todos los pares.
2. Se encuentra el costo máximo $\max_{i \in V} \max_{j \in V} A_{ij}$; esto da la excentricidad del vértice i .

El tiempo de ejecución de este proceso está dominado por el primer paso, que lleva un tiempo $\mathcal{O}(n^3)$. El paso (2) lleva $\mathcal{O}(n^2)$ y el paso (3) lleva $\mathcal{O}(n)$.

Ejemplo 6.12. La matriz de costo CMCP para la figura 6.22 se muestra en la figura 6.23. El valor máximo de cada columna se muestra a continuación.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>a</i>	0	1	3	5	7
<i>b</i>	∞	0	2	4	6
<i>c</i>	∞	3	0	2	4
<i>d</i>	∞	1	3	0	7
<i>e</i>	∞	6	8	5	0
máx	∞	6	8	5	7

Fig. 6.23. Matriz de costo CMCP.

6.5 Recorridos en grafos dirigidos

Para resolver con eficiencia muchos problemas relacionados con grafos dirigidos, es necesario visitar los vértices y los arcos de manera sistemática. La búsqueda en profundidad, una generalización del recorrido en orden previo de un árbol, es una técnica importante para lograrlo, y puede servir como estructura para construir otros algoritmos eficientes. Las dos últimas secciones de este capítulo contienen varios algoritmos que usan esta búsqueda como base.

Supóngase que se tiene un grafo dirigido G en el cual todos los vértices están marcados en principio como *no visitados*. La búsqueda en profundidad trabaja seleccionando un vértice v de G como vértice de partida; v se marca como *visitado*. Después, se recorre cada vértice adyacente a v no visitado, usando recursivamente la búsqueda en profundidad. Una vez que se han visitado todos los vértices que se pueden alcanzar desde v , la búsqueda de v está completa. Si algunos vértices quedan sin visitar, se selecciona alguno de ellos como nuevo vértice de partida, y se repite este proceso hasta que todos los vértices de G se hayan visitado.

Esta técnica se conoce como búsqueda en profundidad porque continúa buscando en la dirección hacia adelante (más profunda) mientras sea posible. Por ejemplo, supóngase que x es el vértice visitado más recientemente. La búsqueda en profundidad selecciona algún arco no explorado $x \rightarrow y$ que parte de x . Si se ha visitado y , el procedimiento intenta continuar por otro arco que no se haya explorado y que parte de x . Si y no se ha visitado, entonces el procedimiento marca y como visitado e inicia una nueva búsqueda a partir de y . Después de completar la búsqueda de todos los caminos que parten de y , la búsqueda regresa a x , el vértice desde el cual se visitó y por primera vez. Se continúa el proceso de selección de arcos sin explorar que parten de x hasta que todos los arcos de x han sido explorados.

Puede usarse una lista de adyacencia $L[v]$ para representar los vértices adyacentes al vértice v , y un arreglo *marca* cuyos elementos son del tipo (*visitado*, *no_visitado*).

tado), puede usarse para determinar si un vértice ya fue visitado antes. El procedimiento recursivo *bpf* se describe en la figura 6.24. Para usarlo en un grafo con n vértices, se asigna el valor inicial *marca* a *no_visitado*, y después se comienza la búsqueda en profundidad con cada vértice que aún permanezca sin visitar cuando llegue su turno, con

```
for  $v := 1$  to  $n$  do
    marca[ $v$ ] := no_visitado;
for  $v := 1$  to  $n$  do
    if marca[ $v$ ] = no_visitado then
        bpf( $v$ )
```

Obsérvese que la figura 6.24 es un modelo al cual se agregarán después otras acciones, al aplicar la búsqueda en profundidad. Lo único que hace el código de la figura 6.24 es actualizar el arreglo *marca*.

Análisis de la búsqueda en profundidad

Todas las llamadas a *bpf* en la búsqueda en profundidad de un grafo con a arcos y $n \leq a$ vértices lleva un tiempo $O(a)$. Para ver por qué, obsérvese que en ningún vértice se llama a *bpf* más de una vez, porque tan pronto como se llama a *bpf*(v) se hace *marca*[v] igual a *visitado* en la línea (1), y nunca se llama a *bpf* en un vértice que antes tenía su *marca* igual a *visitado*. Así, el tiempo total consumido en las líneas (2) y (3) recorriendo las listas de adyacencias es proporcional a la suma de las longitudes de dichas listas, esto es, $O(a)$. De esta forma, suponiendo que $n \leq a$, el tiempo total consumido por la búsqueda en profundidad de un grafo completo es $O(a)$, lo cual es, hasta un factor constante, el tiempo necesario simplemente para recorrer cada arco.

```
procedure bpf(  $v$ : vértice );
var
     $w$ : vértice;
begin
(1)    marca[ $v$ ] := visitado;
(2)    for cada vértice  $w$  en  $L[v]$  do
            if marca[ $w$ ] = no_visitado then
                bpf( $w$ )
end; | bpf
```

Fig. 6.24. Búsqueda en profundidad.

Ejemplo 6.13. Supóngase que el procedimiento *bpf*(v) se aplica al grafo dirigido de la figura 6.25 con $v = A$. El algoritmo marca *A* como visitado y selecciona el vértice *B* en las lista de adyacencia del vértice *A*. Puesto que *B* no se ha visitado, la búsque-

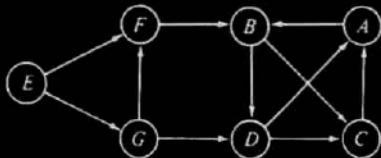


Fig. 6.25. Grafo dirigido.

ciona el primer vértice en la lista de adyacencia del vértice B . Dependiendo del orden de los vértices en la lista de adyacencia de B , la búsqueda seguirá con C o D .

Suponiendo que C aparece antes que D , se invoca a $bpf(C)$. El vértice A está en la lista de adyacencia de C . Sin embargo, en este momento ya se ha visitado A así que la búsqueda queda en C . Como ya se han visitado todos los vértices de la lista de adyacencia en C , la búsqueda regresa a B , desde donde la búsqueda prosigue a D . Los vértices A y C en la lista de adyacencia de D ya fueron visitados, por lo que la búsqueda regresa a B y después a A .

En este punto, la llamada original a $bpf(A)$ ha terminado. Sin embargo, el grafo dirigido no ha sido recorrido en su totalidad; los vértices E , F y G están sin visitar. Para completar la búsqueda, se puede llamar a $bpf(E)$.

Bosque abarcador en profundidad

Durante un recorrido en profundidad de un grafo dirigido, cuando se recorren ciertos arcos, llevan a vértices sin visitar. Los arcos que llevan a vértices nuevos se conocen como *arcos de árbol* y forman un *bosque abarcador en profundidad* para el grafo dirigido dado. Los arcos continuos de la figura 6.26 forman el bosque abarcador en profundidad del grafo dirigido de la figura 6.25. Obsérvese que los arcos de árbol deben formar realmente un bosque, ya que un vértice no puede estar sin visitar cuando se recorren dos arcos diferentes que llevan a él.

Además de los arcos de árbol, existen otros tres tipos de arcos definidos por una búsqueda en profundidad de un grafo dirigido, que se conocen como arcos de retroceso, arcos de avance y arcos cruzados. Un arco como $C \rightarrow A$ se denomina *arco de retroceso*, porque va de un vértice a uno de sus antecesores en el bosque abarcador. Obsérvese que un arco que va de un vértice hacia sí mismo, es un arco de retroceso. Un arco no abarcador que va de un vértice a un descendiente propio se llama *arco de avance*. En la figura 6.25 no hay arcos de este tipo.

Los arcos como $D \rightarrow C$ o $G \rightarrow D$, que van de un vértice a otro que no es antecesor ni descendiente, se conocen como *arcos cruzados*. Obsérvese que todos los arcos cruzados de la figura 6.26 van de derecha a izquierda, en el supuesto de que se agregan hijos al árbol en el orden en que fueron visitados, de izquierda a derecha, y que se agregan árboles nuevos al bosque de izquierda a derecha. Este patrón no es accidental. Si el arco $G \rightarrow D$ hubiera sido $D \rightarrow G$, entonces no se hubiera visitado G durante la búsqueda en D , y al encontrar el arco $D \rightarrow G$, el vértice G se haría descendiente de D , y $D \rightarrow G$ se convertiría en un arco de árbol.

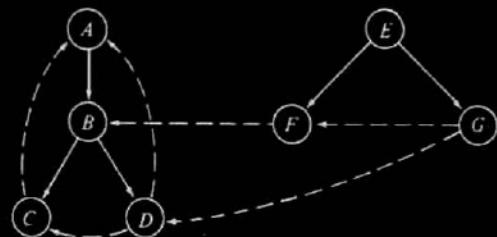


Fig. 6.26. Bosque abarcador en profundidad para la figura 6.25.

¿Cómo distinguir entre los cuatro tipos de arcos? Es obvio que los arcos de árbol son especiales, pues llevan a vértices sin visitar durante la búsqueda en profundidad. Supóngase que se numeran los vértices de un grafo dirigido de acuerdo con el orden en que se marcaron los visitados durante la búsqueda en profundidad. Esto es, se puede asignar a un arreglo:

```
númerop[v] := cont;
cont := cont + 1;
```

después de la línea (1) de la figura 6.24. A esto se le llama *numeración en profundidad* de un grafo dirigido; obsérvese que la numeración en profundidad generaliza la numeración en orden previo introducida en la sección 3.1.

La búsqueda en profundidad asigna a todos los descendientes de un vértice v , números mayores o iguales al número asignado a v . De hecho, w es un descendiente de v si, y sólo si, $\text{númerop}(v) \leq \text{númerop}(w) \leq \text{númerop}(v) + \text{el número de descendientes de } v$. Así, los arcos de avance van de los vértices de baja numeración a los de alta numeración y los arcos de retroceso van de los vértices de alta numeración a los de baja numeración.

Todos los arcos cruzados van de los vértices de alta numeración a los de baja numeración. Para ver esto, supóngase que $x \rightarrow y$ es un arco y $\text{númerop}(x) \leq \text{númerop}(y)$. Así, x se visita antes que y . Todo vértice visitado entre su invocación por primera vez a $bpf(x)$ y el momento en que $bpf(x)$ termina, se convierte en descendiente de x en el bosque abarcador en profundidad. Si y permanece sin visitar cuando se explora el arco $x \rightarrow y$, $x \rightarrow y$ se vuelve un arco de árbol. De otra forma, $x \rightarrow y$ es un arco de avance. Así, $x \rightarrow y$ no puede ser un arco cruzado con $\text{númerop}(x) \leq \text{númerop}(y)$.

En las dos secciones siguientes se analiza la forma de usar la búsqueda en profundidad para la solución de varios problemas de grafos.

6.6 Grafos dirigidos aciclicos

Un *grafo dirigido acíclico*, o *gda*, es un grafo dirigido sin ciclos. Cuantificados en función de las relaciones que representan, los *gda* son más generales que los árboles,

pero menos que los grafos dirigidos arbitrarios. La figura 6.27 muestra un ejemplo de un árbol, un gda, y un grafo dirigido con un ciclo.

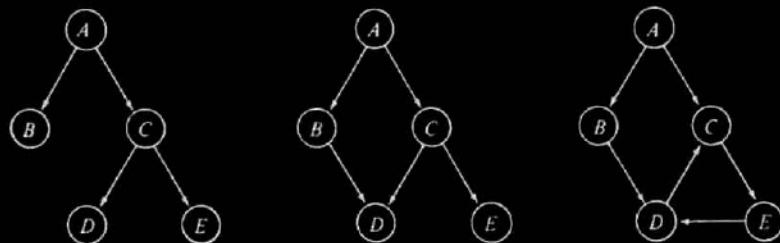


Fig. 6.27. Tres grafos dirigidos.

Entre otras cosas, los gda son útiles para la representación de la estructura sintáctica de expresiones aritméticas con subexpresiones comunes. Por ejemplo, la figura 6.28 muestra un gda para la expresión

$$((a + b)*c + ((a + b)+e)*(e + f)) * ((a + b)*c)$$

Los términos $a + b$ y $(a + b) * c$ son subexpresiones comunes compartidas que se representan con vértices con más de un arco entrante.

Los gda son útiles también para la representación de órdenes parciales. Un *orden parcial R* en un conjunto S es una relación binaria tal que

1. para toda a en S , $a R a$ es falsa (R es irreflexivo), y
2. para toda a, b, c en S , si $a R b$ y $b R c$, entonces $a R c$ (R es transitivo).

Dos ejemplos naturales de órdenes parciales son la relación «menor que» ($<$) en enteros, y la relación de inclusión propia en conjuntos (\subset).

Ejemplo 6.14. Sea $S = \{1, 2, 3\}$ y sea $P(S)$ el conjunto exponencial de S , esto es, el conjunto de todos los subconjuntos de S . $P(S) = [\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}]$. \subset es un orden parcial en $P(S)$. Ciertamente, $A \subset A$ es falso para cualquier conjunto A (irreflexividad), y si $A \subset B$ y $B \subset C$, entonces $A \subset C$ (transitividad). \square

Los gda pueden usarse para reflejar gráficamente órdenes parciales. Para empezar, se puede considerar una relación R como un conjunto de pares (arcos) tales que (a, b) está en el conjunto si, y sólo si, $a R b$ es cierto. Si R es un orden parcial en el conjunto S , entonces el grafo dirigido $G = (S, R)$ es un gda. Del mismo modo, supóngase que $G = (S, R)$ es un gda y R^* es la relación definida por $a R^* b$ si, y sólo si, existe un camino de longitud uno o más que va de a a b . (R^* es la cerradura transitiva de la relación R .) Entonces, R^* es un orden parcial en S .

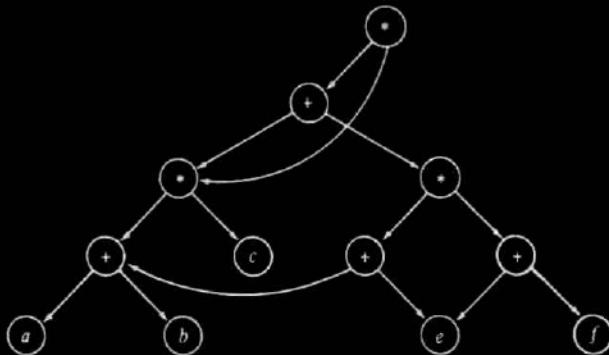


Fig. 6.28. Gda para expresiones aritméticas.

Ejemplo 6.15. La figura 6.29 muestra un gda $(P(S), R)$, donde $S = \{1, 2, 3\}$. La relación R^+ es la inclusión propia en el conjunto exponencial $P(S)$. \square

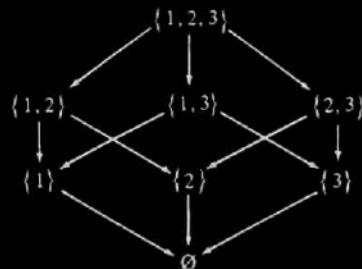


Fig. 6.29. Gda para inclusiones propias.

Prueba de aciclicidad

Se tiene un grafo dirigido $G = (V, A)$, para determinar si G es acíclico, esto es, si G no contiene ciclos. La búsqueda en profundidad puede usarse para responder a esta pregunta. Si se encuentra un arco de retroceso durante la búsqueda en profundidad de G , el grafo tiene un ciclo. Si, al contrario, un grafo dirigido tiene un ciclo, entonces siempre habrá un arco de retroceso en la búsqueda en profundidad del grafo.

Para ver este hecho, supóngase que G es cíclico. Si se efectúa una búsqueda en profundidad en G , habrá un vértice v que tenga el número de búsqueda en profundidad menor en un ciclo. Considérese un arco $u \rightarrow v$ en algún ciclo que contenga

a v . Ya que u está en el ciclo, debe ser un descendiente de v en el bosque abarcador en profundidad. Así, $u \rightarrow v$ no puede ser un arco cruzado. Puesto que el número en profundidad de u es mayor que el de v , $u \rightarrow v$ no puede ser un arco de árbol ni un arco de avance. Así que $u \rightarrow v$ debe ser un arco de retroceso, como se ilustra en la figura 6.30.



Fig. 6.30. Todo ciclo contiene un arco de retroceso.

Clasificación topológica

Un proyecto grande suele dividirse en una colección de tareas más pequeñas, algunas de las cuales se han de realizar en ciertos órdenes específicos, de modo que se pueda culminar el proyecto total. Por ejemplo, una carrera universitaria puede tener cursos que requieran otros como prerrequisitos. Los gda pueden emplearse para modelar de manera natural estas situaciones. Por ejemplo, podría tenerse un arco del curso C al curso D si C fuera un prerrequisito de D .

Ejemplo 6.16. La figura 6.31 muestra un gda con la estructura de prerrequisitos de cinco cursos. El curso C_3 , por ejemplo, requiere los cursos C_1 y C_2 como prerrequisitos. □

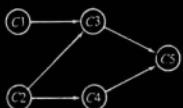


Fig. 6.31. Gda de prerrequisitos.

La *clasificación topológica* es un proceso de asignación de un orden lineal a los vértices de un gda tal que si existe un arco del vértice i al vértice j , i aparece antes que j en el ordenamiento lineal. Por ejemplo, $C1, C2, C3, C4, C5$ es una clasificación topológica del gda de la figura 6.31. Al tomar los cursos en esta secuencia, se puede satisfacer la estructura de prerequisitos dada en la figura.

La clasificación topológica puede efectuarse con facilidad si se agrega una instrucción de impresión después de la línea (4) al procedimiento de búsqueda en profundidad de la figura 6.24:

```
procedure clasificación_topológica(v: vértice);
    {imprime los vértices accesibles desde v en orden topológico invertido}
    var
        w: vértice;
    begin
        marca[v] := visitado;
        for cada vértice w en  $L[v]$  do
            if marca[w] = no_visitado then
                clasificación_topológica(w);
                writeln(v)
        end; {clasificación_topológica}
```

Cuando *clasificación_topológica* termina de buscar en todos los vértices adyacentes a un vértice dado x , imprime x . El efecto de llamar a *clasificación_topológica*(v) es imprimir en orden topológico inverso todos los vértices de un gda accesibles desde v por medio de un camino en el gda.

Esta técnica funciona porque no existen arcos de retroceso en un gda. Considérese lo que sucede cuando la búsqueda en profundidad deja un vértice x por última vez. Los únicos arcos que emanan de v son arcos de árbol, de avance y cruzados. Pero todos esos arcos están dirigidos hacia vértices que ya se han visitado completamente y que, por tanto, preceden a x en el orden que se están construyendo.

6.7 Componentes fuertes

Un componente fuertemente conexo de un grafo dirigido es un conjunto maximal de vértices en el cual existe un camino que va desde cualquier vértice del conjunto hasta cualquier otro vértice también del conjunto. La búsqueda en profundidad puede usarse para determinar con eficiencia los componentes fuertemente conexos de un grafo dirigido.

Sea $G = (V, A)$ un grafo dirigido; se puede dividir V en clases de equivalencia V_i , $1 \leq i \leq r$, tales que los vértices v y w son equivalentes si, y sólo si, existe un camino de v a w y otro de w a v . Sea A_i , $1 \leq i \leq r$, el conjunto de los arcos con cabeza y cola en V_i . Los grafos $G_i = (V_i, A_i)$ se denominan componentes fuertemente conexos (o sólidas componentes fuertes) de G . Un grafo dirigido con sólo un componente fuerte

Ejemplo 6.17. La figura 6.32 ilustra un grafo dirigido con los dos componentes fuertes mostrados en la figura 6.33. \square

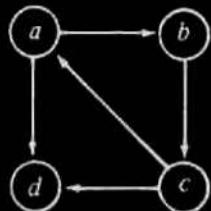


Fig. 6.32. Grafo dirigido.

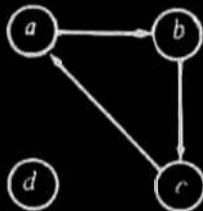
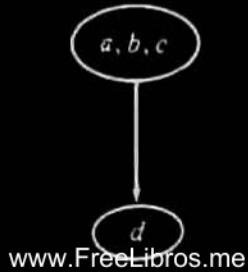


Fig. 6.33. Componentes fuertes del grafo de la figura 6.32.

Obsérvese que todo vértice de un grafo dirigido G está en algún componente fuerte, pero que ciertos arcos pueden no estarlo. Tales arcos, llamados arcos de *cruce de componentes*, van de un vértice de un componente a un vértice de otro. Se pueden representar las interconexiones entre los componentes construyendo un *grafo reducido* de G , cuyos vértices son los componentes fuertemente conexos de G . Hay un arco de un vértice C a un vértice diferente C' de este tipo de grafos, si existe un arco en G que vaya de algún vértice del componente C a algún otro del componente C' . El grafo reducido siempre es un gda, porque si existiera algún ciclo, todos los componentes del *ciclo* serían en realidad un solo componente fuerte, lo cual significaría que no se calcularon en forma adecuada los componentes fuertes. La figura 6.34 muestra el grafo reducido del grafo dirigido de la figura 6.32.



Ahora se presenta un algoritmo para encontrar los componentes fuertemente conexos de un grafo dirigido G dado.

1. Efectúese una búsqueda en profundidad de G y numérense los vértices en el orden de terminación de las llamadas recursivas; esto es, asígnese un número al vértice v después de la línea (4) de la figura 6.24.
2. Constrúyase un grafo dirigido nuevo G_r invirtiendo las direcciones de todos los arcos de G .
3. Realícese una búsqueda en profundidad en G_r , partiendo del vértice con numeración más alta de acuerdo con la numeración asignada en el paso (1). Si la búsqueda en profundidad no llega a todos los vértices, iniciese la siguiente búsqueda a partir del vértice restante con numeración más alta.
4. Cada árbol del bosque abarcador resultante es un componente fuertemente conexo de G .

Ejemplo 6.18. Se aplica este algoritmo al grafo dirigido de la figura 6.32, partiendo de a y continuando primero hasta b . Después del paso (1) se numeran los vértices como se muestra en la figura 6.35. Al invertir la dirección de los arcos, se obtiene el grafo G_r de la figura 6.36.

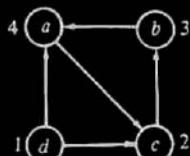
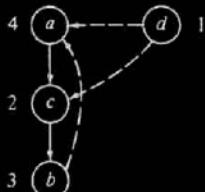
Cuando se realiza la búsqueda en profundidad en G_r , surge el bosque abarcador en profundidad de la figura 6.37. Se comienza con a como raíz, porque a tiene el número más alto. Desde a sólo se alcanza c y después b . El siguiente árbol tiene raíz d , ya que es el siguiente (y único) vértice restante con numeración más alta. Cada árbol del bosque forma un componente fuertemente conexo del grafo dirigido original. \square

Se ha pretendido que los vértices de un componente fuertemente conexo se correspondan con los vértices de un árbol del bosque abarcador de la segunda búsqueda en profundidad. Para ver por qué, obsérvese que si v y w son vértices del mismo componente fuertemente conexo, existen caminos en G desde v hasta w y desde w hasta v . Así, existen también caminos desde v hasta w y desde w hasta v en G_r .

Supóngase que en la búsqueda en profundidad de G , se inicia la búsqueda en alguna raíz x y se llega hasta v o w . Como v y w se alcanzan uno al otro, ambos terminarán formando parte del árbol abarcador con raíz x .



Fig. 6.35. Despues del paso 1.

Fig. 6.36. G_r .Fig. 6.37. Bosque abarcador en profundidad para G_r .

Ahora, supóngase que v y w están en el mismo árbol abarcador del bosque abarcador en profundidad de G_r . Se debe demostrar que v y w están en el mismo componente fuertemente conexo. Sea x la raíz del árbol abarcador que contiene v y w . Puesto que v es descendiente de x , existe un camino en G , que va de x a v . Así, existe un camino en G de v a x .

En la construcción del bosque abarcador en profundidad de G_r , el vértice v quedó sin visitarse cuando se inició la búsqueda en x . De aquí que x tiene un número mayor que v , por lo que en la búsqueda en profundidad de G , la llamada recursiva en v terminó antes que la llamada recursiva en x . Pero en la búsqueda en profundidad de G , la búsqueda en v no pudo haberse iniciado antes que la de x , ya que el camino en G de v a x implicaría que la búsqueda en x empezaría y terminaría antes de terminar la búsqueda en v .

Se concluye que en la búsqueda de G , v se visita durante la búsqueda de x , por lo que, v es descendiente de x en el primer bosque abarcador en profundidad de G . Así, existe un camino de x a v en G . Por tanto, x y v están en el mismo componente fuertemente conexo. Un razonamiento idéntico muestra que x y w están en el mismo componente fuertemente conexo y, por tanto, v y w también lo están, como muestran los caminos que van de v a x a w , y de w a x a v .

Ejercicios

- 6.1 Represéntese el grafo dirigido de la figura 6.38
 - a) por medio de una matriz de adyacencia dando los costos de los arcos, y
 - b) por medio de una lista enlazada de adyacencia con indicación de los costos de los arcos.

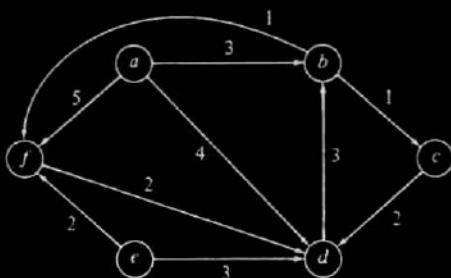


Fig. 6.38. Grafo dirigido con costos de los arcos.

- 6.2 Describase un modelo matemático para el siguiente problema de horarios. Dadas las tareas T_1, T_2, \dots, T_n , que requieren tiempos t_1, t_2, \dots, t_n para ejecutarse, y un conjunto de restricciones, cada una de la forma « T_i debe terminar antes del inicio de T_j », encuéntrese el tiempo mínimo necesario para ejecutar todas las tareas.
- 6.3 Realicense las operaciones PRIMERO, SIGUIENTE y VERTICE para grafos dirigidos representados por
- matrices de adyacencia,
 - listas enlazadas de adyacencias, y
 - listas de adyacencias como la representada en la figura 6.5.
- 6.4 En el grafo dirigido de la figura 6.38,
- empleése el algoritmo *Dijkstra* para encontrar los caminos más cortos que van del vértice a a los otros vértices.
 - utilícese el algoritmo *Floyd* para encontrar las distancias más cortas entre todos los pares de puntos. Constrúyase también la matriz P que permita recuperar los caminos más cortos.
- 6.5 Escribábase un programa completo para el algoritmo de Dijkstra usando árboles parcialmente ordenados como colas de prioridad y listas enlazadas de adyacencias.
- *6.6 Demuéstrese que el programa *Dijkstra* no funciona bien si los arcos tienen costos negativos.
- **6.7 Muéstrese que el programa *Floyd* funciona si alguno de los arcos, pero ningún ciclo, tienen costo negativo.
- 6.8 Suponiendo que el orden de los vértices es a, b, \dots, f en la figura 6.38, constrúyase un bosque abarcador en profundidad; indíquese los arcos de árbol, de retroceso, de avance y cruzados, y la numeración en profundidad de los vértices.

- *6.9 Supóngase que se tiene un bosque abarcador en profundidad, y se lista en orden posterior cada uno de los árboles abarcadores (los árboles que están formados por aristas abarcadoras), de izquierda a derecha. Demuéstrese que este orden es el mismo en el que terminaron las llamadas a *bpf* cuando se construyó el bosque abarcador.
- 6.10 Una *raíz* de un gda es un vértice *r* tal que todo vértice del gda puede alcanzarse por un camino dirigido desde *r*. Escribase un programa para determinar si un gda posee raíz.
- *6.11 Considérese un gda con *a* arcos y con dos vértices distintos *s* y *t*. Constrúyase un algoritmo $O(a)$ para encontrar el conjunto maximal de caminos disjuntos de *s* a *t*. Por maximal se entiende que ya no se pueden añadir caminos adicionales, pero eso no significa que sea el tamaño más grande para ese conjunto.
- 6.12 Constrúyase un algoritmo para convertir un árbol de expresiones con los operadores + y * en un gda al compartir subexpresiones comunes. ¿Cuál es la complejidad de tiempo de ese algoritmo?
- 6.13 Constrúyase un algoritmo para la evaluación de expresiones aritméticas representadas con un gda.
- 6.14 Escribase un programa para encontrar el camino más largo en un gda. ¿Cuál es la complejidad de tiempo de este programa?
- 6.15 Encuéntrense los componentes fuertes de la figura 6.38.
- *6.16 Pruébese que el grafo reducido de los componentes fuertes de la sección 6.7 debe ser un gda.
- 6.17 Dibújese el primer bosque abarcador, el grafo invertido y el segundo bosque abarcador que se obtiene al aplicar el algoritmo de componentes fuertes al grafo dirigido de la figura 6.38.
- 6.18 Obténgase el algoritmo de componentes fuertes analizado en la sección 6.7.
- *6.19 Muéstrese que el algoritmo de componentes fuertes requiere un tiempo $O(a)$ en un grafo dirigido de *a* arcos y *n* vértices, suponiendo que $n \leq a$.
- *6.20 Escribase un programa que tome como entrada un grafo dirigido y dos de sus vértices. El programa debe imprimir todos los caminos simples que vayan de un vértice al otro. ¿Cuál es la complejidad de tiempo de este programa?
- *6.21 Una *reducción transitiva* de un grafo dirigido $G = (V, A)$ es cualquier grafo G' con los mismos vértices pero con la menor cantidad de arcos posible, de modo que el cierre transitivo G' es el mismo que el de G . Demuéstrese que si G es un gda, la reducción transitiva de G es única.

- *6.22 Escribase un programa para obtener la reducción transitiva de un grafo dirigido. ¿Cuál es la complejidad de tiempo de este programa?
- *6.23 $G' = (V, A')$ se conoce como el *grafo dirigido equivalente minimal* de un grafo dirigido $G = (V, A)$, si A' es el subconjunto más pequeño de A y la cerradura transitiva de G y G' es el mismo. Demuéstrese que si G es acíclico, sólo hay un grafo dirigido equivalente minimal, es decir, la reducción transitiva.
- *6.24 Escribase un programa para encontrar un grafo dirigido equivalente minimal para un grafo dirigido dado. ¿Cuál es la complejidad de tiempo de ese programa?
- *6.25 Escribase un programa para encontrar el camino simple más largo de un vértice dado de un grafo dirigido. ¿Cuál es la complejidad de tiempo del programa?

Notas bibliográficas

Berge [1985] y Harary [1969] son dos buenas fuentes de material suplementario sobre teoría de grafos. Algunos libros que tratan algoritmos sobre grafos son Deo [1975], Even [1980] y Tarjan [1983].

El algoritmo para caminos más cortos con un solo origen de la sección 6.3 se debe a Dijkstra [1959]. El algoritmo de los caminos más cortos entre todos los pares es de Floyd [1962] y el de cerradura transitiva es de Warshall [1962]. Johnson [1977] analiza algoritmos eficientes para encontrar caminos más cortos en grafos dispersos. Knuth [1968] contiene material adicional sobre clasificación topológica.

El algoritmo de componentes fuertes de la sección 6.7 es similar al sugerido por R. Kosaraju en 1978 (sin publicar), y al publicado por Sharir [1981]. Tarjan [1972] contiene otro algoritmo de componentes fuertes que sólo necesita un recorrido con búsqueda en profundidad.

Coffman [1976] contiene muchos ejemplos de cómo se pueden usar los grafos dirigidos para los problemas de modelado de horarios, como en el ejercicio 6.2. Aho, Garey y Ullman [1972] muestran que la reducción transitiva de un gda es única, y que el cálculo de la reducción transitiva de un grafo dirigido es, computacionalmente, equivalente al cálculo de la cerradura transitiva (Ejercicios 6.21 y 6.22). La obtención del *grafo dirigido equivalente minimal* (Ejercicios 6.23 y 6.24), por otro lado, parece ser mucho más difícil desde el punto de vista computacional; este problema es NP-completo [Sahni (1974)].